holisticon

Camunda BPM Data

Beautiful process data handling for Camunda BPM

Some history

- Around 10 years ago Bernd Ruecker started the first Camunda community skype call
- We arranged to meet regularily
- Martin Schimak, Bernd, Jan and me were there (and some guys I don't remember)
- The second meetup was dedicated to data in processes

Camunda Best Practices

Changelog

July 27, 2016 - Initial Release of Camunda Best Practices

27 best practice articles, all of them being focused on concrete practical challenges.

- **New Content**

  Building a Custom Tasklist/Application
  Cleaning Up Historical Data
  Creating Readable Process Models
  Dealing With Problems and Exceptions
  Deciding About Your Stack
  Deciding About Your Tasklist
  Handling Data in Processes
  Enhancing Tasklists with Business Data
  Estimating Effort
  Following the Customer Success Path
  Integrating a Third Party Tasklist
  Invoking Services from the Process

Process Variables in Camunda BPM

holisticon

Basic Idea

- *Dynamic* map of process variables
- In every process step this map can be accessed and modified
- Changes in this map corresponds to *data flow*
- The `key` in this map corresponds to process variable name (`String`)
- The `value` reflects the process variable value (`Object`)

Process data loading strategies

- full
  - load all data and store it in process variables
  - all data available for delegates
  - snapshot of the data
- references only
  - work only with references
  - data must be loaded via reference
  - data is in-sync

> *Camunda says: "As a rule of thumb, store as less few variables as possible within Camunda."*

Process data storage strategies

- flat variables
  - many variables
  - only some pre-defined types
  - no serialization issues
- rich objects
  - fewer variables
  - serialization matters

Process data serialization

- basic types
  - Number
  - String
  - Boolean
  - Date
  - bytes
  - file
- binary (Java Serializable)
- SPIN
  - XML
  - JSON (via Jackson)

Thank you for listening... you made it!

Wait, there is more...

Using **Constants** and Data **Accessors**

Avoid the copy/paste of string representations of your process variable names across your code base. **At least** collect the variable names for a process definition in a **Constants** interface/class:

```
public interface TwitterDemoProcessConstants {
  String VAR_NAME_TWEET = "tweet";
  String VAR_NAME_APPROVED = "approved";
}
```

This way, you have much more security against typos and can easily make use of refactoring mechanisms offered by IDEs.

However, if you also want to avoid spreading the necessary **type conversion (casting)** all over your application and want to have a natural place for **serializing your complex process variables**, we recommend that you use a **Data Accessor** class. It comes in two flavors:

- A **Process Data Accessor**: knows the names and types of all process variables of a certain process definition. It serves as the central point to declare variables for that process.
- A **Process Variable Accessor**: encapsulates the access to exactly one variable. This is useful if you reuse certain variables in different processes.

What does it mean?

```
/**
 * Constants
 */
public class OrderProcessConstants {
  public static final String ORDER = "order";
  public static final String ORDER_APPROVED = "orderApproved";
}

/**
 * Data Accessor
 */
public class OrderProcessVariables {

  private VariableScope variableScope;

  public OrderProcessVariables(VariableScope variableScope) {
    this.variableScope = variableScope;
  }


  public Order getOrder() {
    return (Order) variableScope.getVariable(OrderProcessConstants.ORDER);
  }

  public void setOrder(Order order) {
    variableScope.setVariable(OrderProcessConstants.ORDER, order);
  }

  public Boolean isApproved() {
    return (Boolean) variableScope.getVariable(OrderProcessConstants.ORDER_APPROVED);
  }

  public void setApproved(Boolean approved) {
    variableScope.setVariable(OrderProcessConstants.ORDER_APPROVED, approved);
  }
}
```

Room for improvement

- Constants hold variable names
- Type must be held in the data accessor
- Data accessor for multiple scopes (delegate, services)
- An implicit type downcast is required (or the client must know the type)
- Pretty verbose
- Plumbing in business code...
- Testing of Camunda services is silly

Time for a new library!

- FOSS library for working with Camunda variables
- Variable Factory (data accessor)
- improved API (fluent, more intuitive)
- guarantees type-safety
- guards for validation
- building blocks for anti-corruption-layer
- Kotlin extension functions
- Mockito support methods for testing

Variable declaration example

```
import io.holunda.camunda.bpm.data.factory.VariableFactory;
import static io.holunda.camunda.bpm.data.CamundaBpmData.*;

public class OrderApproval {

  public static final VariableFactory<String>              ORDER_ID = stringVariable("orderId");

  public static final VariableFactory<Order>               ORDER = customVariable("order", Order.class);
  public static final VariableFactory<Integer>             ORDER_POSITIONS_COUNT = integerVariable("orderPositionsCount");
  public static final VariableFactory<List<OrderPosition>> ORDER_POSITIONS = listVariable("orderPositions", OrderPosition.class);

}
```

Variable access

```
public class MyDelegate implements JavaDelegate {
  public void execute(DelegateExecution execution) {
    Order order = ORDER.from(execution).get(); // no downcast needed
  }
}
public class TaskHelper {

  @Autowired
  private TaskService taskService;

  public void setNewValuesForTask(String taskId, String orderId, Boolean orderApproved) {
      ORDER_ID.on(taskService, taskId).set(orderId);
      ORDER_APPROVED.on(taskService, taskId).setLocal(orderApproved);
  }
}
```

Summary and Outlook

- provides facilities to implement **Camunda Best Practices**
- allows for **type safe** access to process variables by
  - variable factories
  - variable readers / writers (Fluent API)
  - variable guards
  - anti-corruption-layer
  - testing support for Camunda Services access
- it is 100% open source
  - APACHE 2.0 License
  - released on Maven Central (`io.holunda.data:camunda-bpm-data`)
  - Contributions are welcome
- next steps
  - will move to Camunda Community HUB soon
  - get closer to Camunda BPM Mockito

References

- https://github.com/holunda-io/camunda-bpm-data
- https://www.holunda.io/camunda-bpm-data/
- https://camunda.com/best-practices/handling-data-in-processes/
- https://github.com/camunda/camunda-bpm-mockito